

# 浙江大学

## 本科实验报告

课程名称：计算机体系结构

姓 名：范源颢

学 院：计算机科学与技术学院

系：计算机科学与技术系

专 业：计算机科学与技术

学 号：3180103574

指导教师：陈文智

2020 年 12 月 22 日

# 浙江大学实验报告

课程名称： 计算机体系结构 实验类型： 综合

实验项目名称： 31 指令流水线 CPU 设计

学生姓名： 范源颢 专业： 计算机科学技术 学号： 3180103574

同组学生姓名： 周寒靖 指导老师： 陈文智

实验地点： 曹光彪西楼-301 实验日期： 2020 年 12 月 22 日

## 一、 实验目的和要求

目的：

1. 理解 MIPS 的 31 条指令
2. 掌握可执行 31 条指令的流水线 CPU 的设计方法
3. 掌握 31 指令流水线 CPU 的程序验证方法

要求：

1. 设计流水线 CPU，使得它支持执行 31 条 MIPS 指令，包括设计数据通路 datapath 和 CPU 控制器 CPU controller
2. 用程序验证单周期 CPU，观察程序的执行情况。

## 二、 实验内容和原理

- 2.1 我们在之前的试验中仅仅实现 16 指令的 MIPS 汇编语言，这一次实验我们需要将支持的 MIPS 指令扩展为 31 条，支持的指令集扩大了一倍。我们将新旧的指令集展示如下：

# 16 MIPS Instructions



Bit #	31..26	25..21	20..16	15..11	10..6	5..0	Operations
R-type	op	rs	rt	rd	sa	func	
add		rs	rt	rd	00000	100000	rd = rs + rt; with overflow PC+=4
sub		rs	rt	rd	00000	100010	rd = rs - rt; with overflow PC+=4
and		rs	rt	rd	00000	100100	rd = rs & rt; PC+=4
or		rs	rt	rd	00000	100101	rd = rs   rt; PC+=4
sll	000000	00000	rt	rd	sa	000000	rd = rt << sa; PC+=4
srl		00000	rt	rd	sa	000010	rd = rt >> sa (logical); PC+=4
slt		rs	rt	rd	00000	101010	if(rs < rt)rd = 1; else rd = 0; <(signed) PC+=4
jr		rs	00000	00000	00000	001000	PC=rs PC+=4
I-type	op	rs	rt		immediate		
addi	001000	rs	rt		imm		rt = rs + (sign_extend)imm; with overflow PC+=4
andi	001100	rs	rt		imm		rt = rs & (zero_extend)imm; PC+=4
ori	001101	rs	rt		imm		rt = rs   (zero_extend)imm; PC+=4
lw	100011	rs	rt		imm		rt = memory[rs + (sign_extend)imm]; PC+=4
sw	101011	rs	rt		imm		memory[rs + (sign_extend)imm] <-- rt; PC+=4
beq	000100	rs	rt		imm		if (rs == rt) PC+=4 + (sign_extend)imm <<2; PC+=4
J-type	op			address			
j	000010			address			PC = (PC+4)[31..28],address<<2
jal	000011			address			PC = (PC+4)[31..28],address<<2 ; \$31 = PC+4

旧指令集 ↑

## MIPS Instructions



Bit #	31..26	25..21	20..16	15..11	10..6	5..0	Operations
R-type	op	rs	rt	rd	sa	func	
add		rs	rt	rd	00000	100000	rd = rs + rt; with overflow PC+=4
addu		rs	rt	rd	00000	100001	rd = rs + rt; without overflow PC+=4
sub		rs	rt	rd	00000	100010	rd = rs - rt; with overflow PC+=4
subu		rs	rt	rd	00000	100011	rd = rs - rt; without overflow PC+=4
and		rs	rt	rd	00000	100100	rd = rs & rt; PC+=4
or		rs	rt	rd	00000	100101	rd = rs   rt; PC+=4
xor		rs	rt	rd	00000	100110	rd = rs ^ rt; PC+=4
nor		rs	rt	rd	00000	100111	rd = ~(rs   rt); PC+=4
slt	000000	rs	rt	rd	00000	101010	if(rs < rt)rd = 1; else rd = 0; <(signed) PC+=4
sltu		rs	rt	rd	00000	101011	if(rs < rt)rd = 1; else rd = 0; <(unsigned) PC+=4
sll		00000	rt	rd	sa	000000	rd = rt << sa; PC+=4
srl		00000	rt	rd	sa	000010	rd = rt >> sa (logical); PC+=4
sra		00000	rt	rd	sa	000011	rd = rt >> sa (arithmetic); PC+=4
sllv		rs	rt	rd	00000	000100	rd = rt << rs; PC+=4
srlv		rs	rt	rd	00000	000110	rd = rt >> rs (logical); PC+=4
sra v		rs	rt	rd	00000	000111	rd = rt >> rs(arithmetic); PC+=4
jr		rs	00000	00000	00000	001000	PC=rs PC+=4

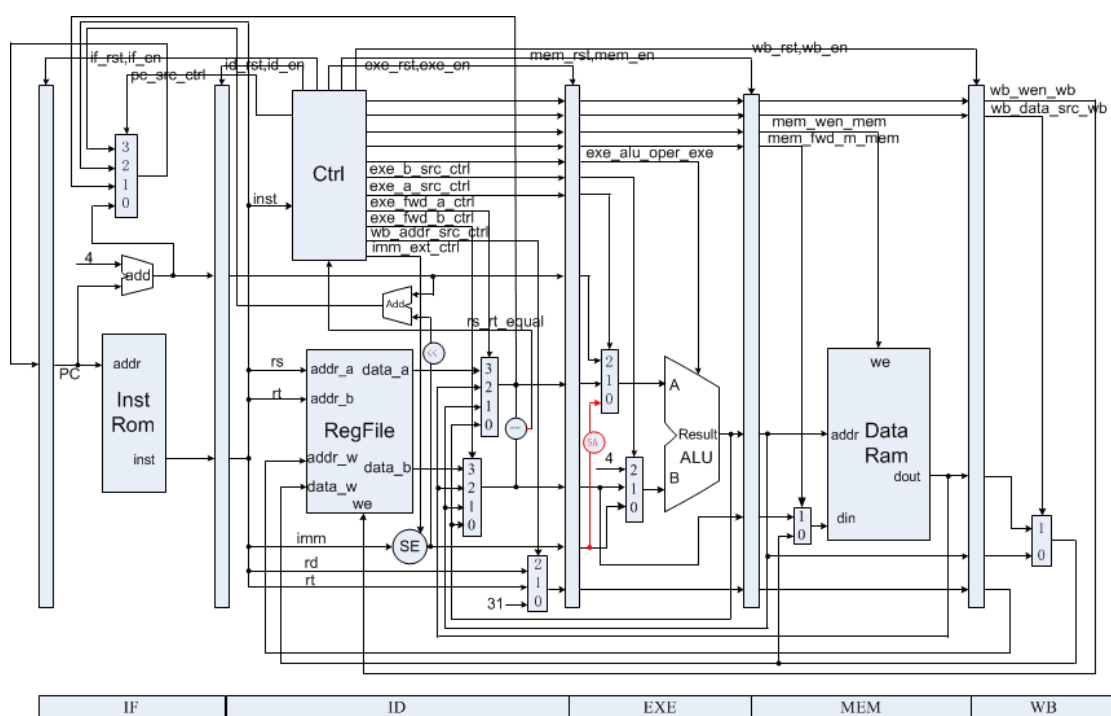
新指令集 1 ↑

MIPS Instructions

Bit #	31..26	25..21	20..16	15..11	10..6	5..0	Operations
I-type	op	rs	rt	immediate			
<u>addi</u>	001000	<u>rs</u>	<u>rt</u>	<u>imm</u>			<u>rt</u> = <u>rs</u> + ( <u>sign_extend</u> ) <u>imm</u> ; with overflow PC+=4
<u>addiu</u>	001001	<u>rs</u>	<u>rt</u>	<u>imm</u>			<u>rt</u> = <u>rs</u> + ( <u>sign_extend</u> ) <u>imm</u> ;without overflow PC+=4
<u>andi</u>	001100	<u>rs</u>	<u>rt</u>	<u>imm</u>			<u>rt</u> = <u>rs</u> & ( <u>zero_extend</u> ) <u>imm</u> ; PC+=4
<u>ori</u>	001101	<u>rs</u>	<u>rt</u>	<u>imm</u>			<u>rt</u> = <u>rs</u>   ( <u>zero_extend</u> ) <u>imm</u> ; PC+=4
<u>xori</u>	001110	<u>rs</u>	<u>rt</u>	<u>imm</u>			<u>rt</u> = <u>rs</u> ^ ( <u>zero_extend</u> ) <u>imm</u> ; PC+=4
<u>lui</u>	001111	00000	<u>rt</u>	<u>imm</u>			<u>rt</u> = <u>imm</u> * 65536; PC+=4
<u>lw</u>	100011	<u>rs</u>	<u>rt</u>	<u>imm</u>			<u>rt</u> = memory[ <u>rs</u> + ( <u>sign_extend</u> ) <u>imm</u> ]; PC+=4
<u>sw</u>	101011	<u>rs</u>	<u>rt</u>	<u>imm</u>			memory[ <u>rs</u> + ( <u>sign_extend</u> ) <u>imm</u> ] <-- <u>rt</u> ; PC+=4
<u>beq</u>	000100	<u>rs</u>	<u>rt</u>	<u>imm</u>			if ( <u>rs</u> == <u>rt</u> ) PC+=4 + ( <u>sign_extend</u> ) <u>imm</u> <<2; PC+=4
<u>bne</u>	000101	<u>rs</u>	<u>rt</u>	<u>imm</u>			if ( <u>rs</u> != <u>rt</u> ) PC+=4 + ( <u>sign_extend</u> ) <u>imm</u> <<2; PC+=4
<u>slti</u>	001010	<u>rs</u>	<u>rt</u>	<u>imm</u>			if ( <u>rs</u> < ( <u>sign_extend</u> ) <u>imm</u> ) <u>rt</u> = 1 else <u>rt</u> = 0; less than signed PC+=4
<u>sltiu</u>	001011	<u>rs</u>	<u>rt</u>	<u>imm</u>			if ( <u>rs</u> < ( <u>zero_extend</u> ) <u>imm</u> ) <u>rt</u> = 1 else <u>rt</u> = 0; less than unsigned PC+=4
J-type	op	address					
<u>j</u>	000010	address					PC = (PC+4)[31..28],address<<2
<u>jal</u>	000011	address					PC = (PC+4)[31..28],address<<2 ; \$31 = PC+4

新指令集 2 ↑

2.2 实验设计的电路图如下所示：



和上一次实验相比增加的就是一条表征偏移指令偏移量的线路 `sa`。

关于偏移指令，`srl`, `sra`, `sll`, 全称分别为

`shift right logical`, 表示逻辑右移, `rt` 中的寄存器右移 `sa` 位赋值给 `rd`, 空余出来的位置补 0;

shift right arithmetical, 表示算数右移, rt 中的寄存器右移 sa 位赋值给 rd, 若原先 rt 首位为 1, 则空余出来的位置补 1, 否则补 0, 这个操作的意图是保留偏移操作之后寄存器数的符号;

shift left logical, 表示逻辑左移, rt 中的寄存器右移 sa 位赋值给 rd, 空位一律补 0。可以看到, 新实验中我们新增了偏移的符号, 这也要求 controller 和 ALU 有需要修改的地方。

其余一些新增的指令包括 addu、subu, addiu, 等等, 这写指令, 显然, 是按照对应的算数操作 add、sub、addi 之后加上 u 的命名规则命名的, 其意义是将相应算数的操作数全部看成无符号数, 我们设计的 CPU 使用补码表示负数, 那么 1 开头的二进制码都会按照符号数进行运算, 我们使用 u 表示不将他们看做负数, 依旧表示较大的正数, 不过, 我们注意到, 这样的数在绝大部分运算(加法 add, 减法 sub, 等等)的过程中和有符号数的运算过程是一样的, 是之后的程序会对这些二进制码做重新解释, 所以我们其实不需要做太多改动, 唯一有区别的是 slt 和 sltu, 我们需要在比较大小的时候重新解释, 在无符号比较时, 1 开头的数码是最大的, 而在有符号比较时, 1 开头的反而是最小的(因为表示负数)。

最后一条需要描述的新指令是 lui, 亦即 Load Unsigned Immediate, 语法“lui rt imm”表示将一个立即数位移之后直接存入到 rt, 我们在 ALU 中新插入操作 lui 进行这条指令的位移, 作为这条指令的 EXE 阶段行为, 之后的 MEM 和 WB 阶段和其他算数指令相同。

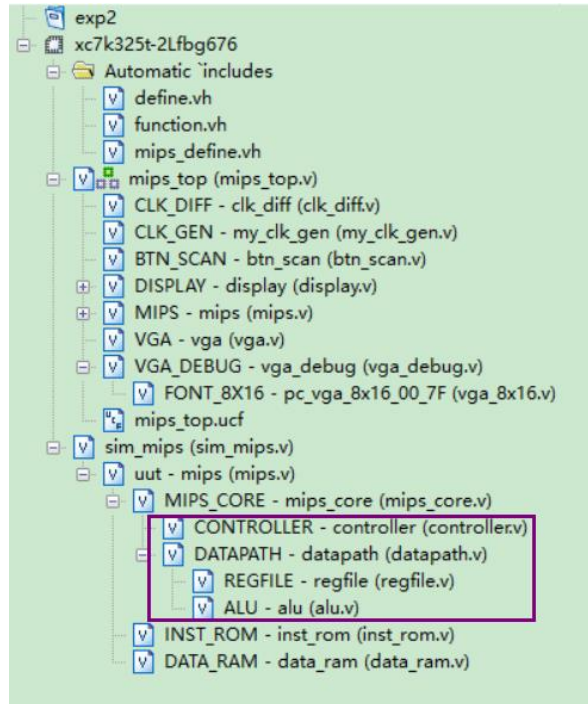
### 2.3 我们的主要工作:

1. 处理 CPU Controller (我们需要根据新指令的 opcode 新增信号, 主要是 sign 指示数据通路是否要考虑符号, 另外处理 lui)
2. 处理 CPU Datapath (我们需要处理 Controller 发出的信号)
3. 处理 mips\_core.v, 将 Controller 的输出和 Datapath 的输入连接起来
4. 处理 ALU 文件, 新增无符号比较和 Lui 的操作符。
5. 对于新增信号的判断, 我们如果想要引入宏增强可读性, 那么我们需要修改 mips\_define.vh
6. 我们需要处理 data\_mem.hex 和 inst\_mem.hex, 修改测试程序, 使得我们的新的

指令能被测试到。

### 三、实验过程和数据记录

实验的文件结构如下：



Automatic includes 文件夹下定义了我们可以在代码中使用的头文件：

mips\_top.v 是实验的主要工程，实现了对于 MIPS 指令的模拟，clk\_diff 和 clk\_gen 模块用于 CPU 内置时钟的信号生成的处理，便于各个进程的同步。Btn\_san 是处理输入信号的模块，时限完成的 mips\_top 模块将烧录到实验室的 sword 开发板上，通过按钮的控制反映 CPU 的计算情况，信息通过 3 个模块输出，分别是 DISPLAY 模块和 VGA，VGA\_DEBUG 模块，mips\_top.ucf 规定了电路的引脚，使得烧录之后的程序可以嵌入开发板；

sim\_mi.v 实际上是一个测试用的工程文件，用于在开发者的电脑上，用 Xilinx ISE 仿真烧录之后的输入输出过程，uut 文件指定了我们的模拟方式（通过内置时钟不断读取 inst\_rom 和 data\_ram 中的信息，并且把相关的信号反馈到电路图上），仿真检测的对象自然是 mips\_core，这个 mips\_core 核上文中 mips\_top 模块中的 mips\_core 是同一个文件。

我们需要完成 mips\_core 中的代码补全。mips.v 文件划分为 MIPS\_CORE 和 inst\_rom，data\_ram，完成了二者（CPU 和内存）的交互，mips\_core.v 划分为 controller.v 和 datapath.v，完成了二者（控制器和数据通路）的交互，我们重点需要完成 controller.v 和 datapath.v 的代码补全。

## 1. CPU datapath 和 mips\_core:

如上文所述，我们首先需要修改 mips\_core，使得 datapath 和 controller 能够交换关于指令是否涉及符号计算的信号，我们将这个信号用 sign 线路传输，这时需要在 datapath 和 controller 中新增端口，我们也命名为.sign，并且通过注释表明这时第五次实验新加的，保存版本信息。如下：

```
//port added in exp5:  
.sign(sign)
```

datapath 是相对比较简单，我们基本上只需要新增 sign 端口，然后将这个端口的信号传递给 ALU 子部件即可，ALU 会根据情况进行恰当的操作符选择，因此，本次实验实际上是 controller 在和 ALU 通信，而其他部件都是通信的接口而已。

下面展示 datapath 中对 ALU 子部件的应用，

```
alu ALU (  
    .a(opa_exe),  
    .b(opb_exe),  
    .oper(exe_alu_oper_exe),  
    .result(alu_out_exe),  
    .sign(alu_sign_exe)////  
);
```

注意这里的 alu\_sign\_exe，我们并不能直接将 sign 信号直接拿来使用，毕竟这个是在 ID 阶段产生的信号，而其使用则是在 EXE 过程，因此我们需要像其他信号一样对此进行按时钟沿传递，也就是在 EXE 阶段的正时钟沿进行赋值：

```
alu_sign_exe<=alu_sign;
```

## 2. CPU Controller 和 mips\_define

我们对于 controller 的内容修改时最重要的，因为 controller 是完成指令解码的主要部件，我们需要在解码时判断指令的操作码（opcode）是否和我们预定义的宏一致，如果是，我们需要根据不同的宏给数据通路发送信号，大部分需要增加的信号都和之前的算数指令信号一致，但注意要新增加 sign 信号

```
case (inst[31:26])  
    INST_R: begin  
        case (inst[5:0])  
            . . . . .  
            //-----new operation-----  
            R_FUNC_ADDU:begin////  
                exe_alu_oper = EXE_ALU_ADD;////
```

```

        wb_addr_src = WB_ADDR_RD;////
        wb_data_src = WB_DATA_ALU;////
        wb_wen = 1;////
        rs_used = 1;////
        rt_used = 1;////
        sign=0;////unsigned
    end
R_FUNC_SUBU:begin
    exe_alu_oper = EXE_ALU_SUB;////
    wb_addr_src = WB_ADDR_RD;////
    wb_data_src = WB_DATA_ALU;////
    wb_wen = 1;////
    rs_used = 1;////
    rt_used = 1;////
    sign=0;////unsigned
end
R_FUNC_XOR:begin
    . . . . .
end
R_FUNC_NOR:begin
    . . . . .
end
R_FUNC_SLTU:begin//sktu $rs $rt $rd #:if(rs < rt)rd
= 1;else rd=0;(< unsigned)
    exe_alu_oper = EXE_ALU_SLT;////new ALU option
    sign = 0;////??reference code fills "1" here
    wb_addr_src = WB_ADDR_RD;////
    wb_data_src = WB_DATA_ALU;////
    wb_wen = 1;////
    rs_used = 1;////
    rt_used = 1;////
end
R_FUNC_SLL: begin////unsigned shift left, SLL $rt,$
rd,sa # rd = rt << sa
    exe_alu_oper = EXE_ALU_SL;////new ALU option
    wb_addr_src = WB_ADDR_RD;////
    wb_data_src = WB_DATA_ALU;////
    exe_a_src = EXE_A_SA;////
    wb_wen = 1;////
    rt_used = 1;////
    sign = 0;////unsigned
end
R_FUNC_SRL: begin////unsigned shift right, SRL $rt,$
rd,sa # rd = rt >> sa (logical)

```



```

        exe_alu_oper = EXE_ALU_SR;////new ALU option
        wb_addr_src = WB_ADDR_RD;////
        wb_data_src = WB_DATA_ALU;////
        exe_a_src = EXE_A_SA;////
        wb_wen = 1;////
        rt_used = 1;////
        sign = 0;////unsigned
    end
    R_FUNC_SRA: begin////signed shift right, SRA $rt,$rd
,sa # rd = rt >> sa (arithmetic)
        exe_alu_oper = EXE_ALU_SR;////
        wb_addr_src = WB_ADDR_RD;////
        wb_data_src = WB_DATA_ALU;////
        exe_a_src = EXE_A_SA;////
        wb_wen = 1;////
        rt_used = 1;////
        sign = 1;////signed
    end
    R_FUNC_SLLV: begin////unsigned shift left, SLLV $rs,
$rt,$rd # rd = rt << rs
        exe_alu_oper = EXE_ALU_SL;//// mistake
        wb_addr_src = WB_ADDR_RD;////
        wb_data_src = WB_DATA_ALU;////
        wb_wen = 1;////
        rs_used = 1;////
        rt_used = 1;////
        sign=0;////unsigned
    end
    R_FUNC_SRLV: begin////unsigned shift right, SRLV $r
s,$rt,$rd # rd= rt >> rs (logical)
        exe_alu_oper = EXE_ALU_SR;////
        wb_addr_src = WB_ADDR_RD;////
        wb_data_src = WB_DATA_ALU;////
        wb_wen = 1;////
        rs_used = 1;////
        rt_used = 1;////
        sign = 0;////unsigned
    end
    R_FUNC_SRAV: begin////signed shift right, SRAV $rs,$r
t,$rd # rd = rt >> rs (arithmetic)
        exe_alu_oper = EXE_ALU_SR;////
        wb_addr_src = WB_ADDR_RD;////
        wb_data_src = WB_DATA_ALU;////
        wb_wen = 1;////

```

```

        rs_used = 1;////
        rt_used = 1;////
        sign = 1;////signed
    end
    //-----
    default: begin
        unrecognized = 1;
    end
endcase

```

上文展示了所有新增加的 Rtype 代码，我们使用宏来表示比较，这些宏定义在 mips\_include 里面，展开之后是操作符对应的二进制码，代码的重用度非常之高，这些指令的代码都是非常类似的，我们在每种指令的开头给出了这种指令的语法。

```

localparam // bit 31:26 for instruction type
    INST_R          = 6'b000000, // bit 5:0 for function type
    R_FUNC_SLL      = 6'b000000,
    R_FUNC_SRL      = 6'b000010, // including ROTR(set bit 21)
    R_FUNC_SRA      = 6'b000011,
    R_FUNC_SLLV     = 6'b000100,
    R_FUNC_SRLV     = 6'b000110, // including ROTRV(set bit 6)
    R_FUNC_SRAV     = 6'b000111,
    R_FUNC_JR       = 6'b001000,

```

上文给出了 mips\_include 中出现的宏。

对于 I-type 的新增指令，我们不在给出完整的代码，只是以 addiu 和 lui 为例展示：

```

INST_ADDIU:begin
    imm_ext = 1;////
    exe_b_src = EXE_B_IMM;////
    exe_alu_oper = EXE_ALU_ADD;////
    wb_addr_src = WB_ADDR_RT;////
    wb_data_src = WB_DATA_ALU;////
    wb_wen = 1;////
    rs_used = 1;////
    sign=0;////unsigned
end
INST_XORI: begin
    . . . . .
end
INST_SLTI: begin
    . . . . .
end
INST_SLTIU: begin
    . . . . .

```

```

end
INST_LUI: begin
    exe_b_src = EXE_B_IMM;////
    exe_alu_oper = EXE_ALU_LUI;////
    wb_wen = 1;////
    rt_used = 1;////
    wb_wen = 1;////
    wb_addr_src = WB_ADDR_RT;////
    wb_data_src = WB_DATA_ALU;////
end

```

### 3. CPU ALU

如前所述，我们需要给 ALU 进行扩展，实质上主要增加的是对 lui 指令的支持

```

EXE_ALU_LUI: begin
    result = {b[15:0], 16'b0};
end

```

对位移操作的指令的支持，通过 sign 信号判断是否需要考虑符号：

```

EXE_ALU_SR: begin
    if(sign)
        result = $signed(b) >>> a; //arithmetic
    else
        result = $unsigned(b) >> a; //logic
    end
end

```

以及扩展比较操作的指令，使之支持无符号数据的比较，简言之，也是通过 sign 增加分支判断

```

EXE_ALU_SLT: begin
    if (sign)
        result = $signed(a) < $signed(b);
    else
        result = $unsigned(a) < $unsigned(b);
    end
end

```

其他指令不用考虑符号，有符号或者无符号都是用同样的算法进行计算的。

我们需要注意我们这里展示的语句全部内嵌在代码块：

```

case (oper)
    EXE_ALU_ADD: begin
        result = a + b;
    end
    . . . . .
endcase

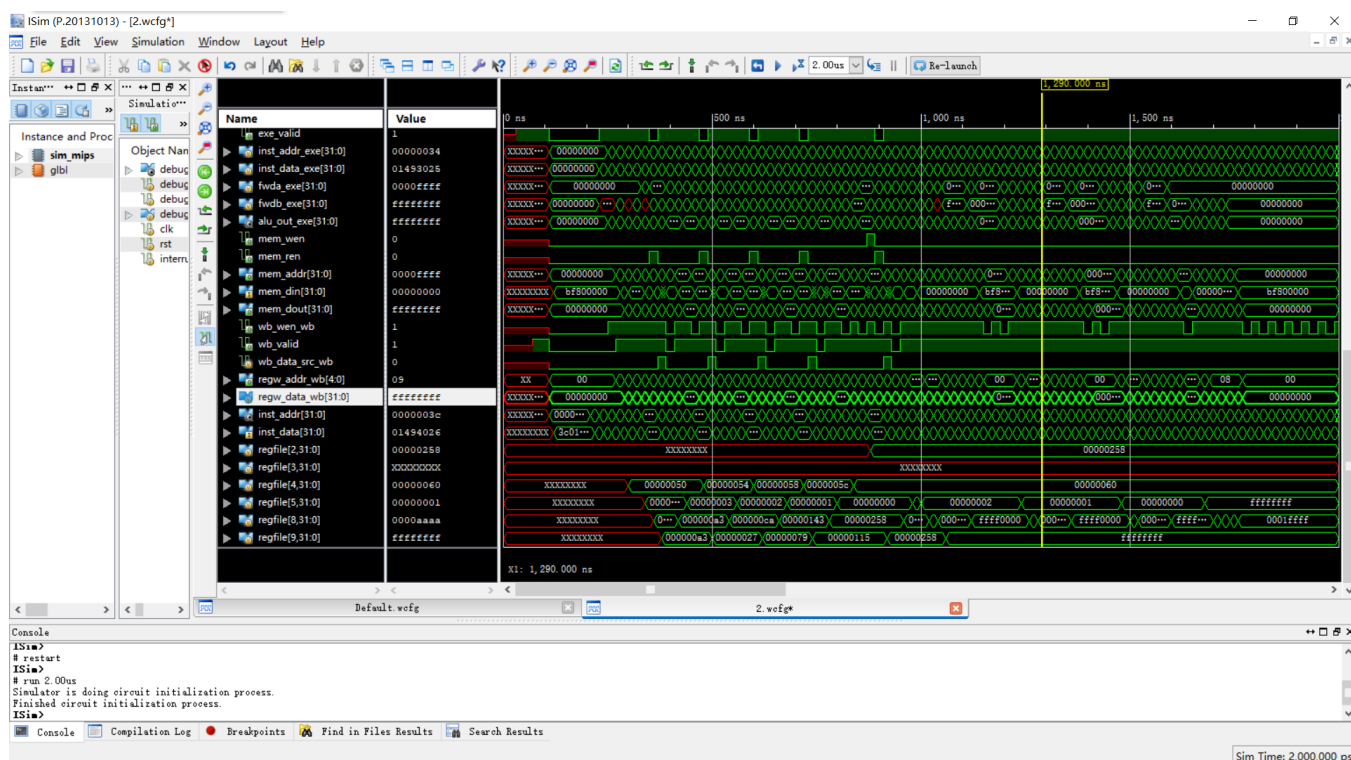
```

中，因此我们的 `oper` 也应该是宏，这些宏在 `controller` 中给出，在 `ALU` 中处理，本质是二进制串的标识符，我们使用宏定义使得代码的可读性增强，为了使得宏定义可以被实现，我们应当在 `mips_define` 中进行定义

```
// EXE ALU operations
localparam
    EXE_ALU_ADD    = 0,
    EXE_ALU_SUB    = 1,
    EXE_ALU_SLT    = 2,
    EXE_ALU_LUI    = 3,
    EXE_ALU_AND    = 4,
    EXE_ALU_OR     = 5,
    EXE_ALU_XOR    = 6,
    EXE_ALU_NOR    = 7,
    EXE_ALU_SL     = 8,
    EXE_ALU_SR     = 9;
```

## 四、实验结果分析

### 4.1 仿真结果：



我们在图片右侧的紫框中找到相应的芯片元件，拖入左侧的图中就可以得到输入输出的信号波形图，注意设定仿真时长，并且显示方式为 16 进制(便于阅读)。

主要观察的波形是 regfile 和 inst 信号的输出,通过观察 inst 信号我们可以确定跳转的指令是否正确,通过观察 regfile 我们可以确定写回的信号是否正确。

#### 4.2 测试用程序介绍

我们测试用的程序如下 (注意和之前的程序不同):

3c010000	% (00)	main:	lui \$1, 0 # address of data[0] %
34240050	% (04)		ori \$4, \$1, 80 # address of data[0] %
0c00001b	% (08)	call:	jal sum # call function %
20050004	% (0c)	ds1ot1:	addi \$5, \$0, 4 # counter,DELYED SLOT(DS) %
ac820000	% (10)	return:	sw \$2, 0(\$4) # store result %
8c890000	% (14)		lw \$9, 0(\$4) # check sw %
01244022	% (18)		sub \$8, \$9, \$4 # sub: \$8 <- \$9 - \$4 %
20050003	% (1c)		addi \$5, \$0, 3 # counter %
20a5ffff	% (20)	loop2:	addi \$5, \$5, -1 # counter - 1 %
34a8ffff	% (24)		ori \$8, \$5, 0xffff # zero-extend: 0000ffff %
39085555	% (28)		xori \$8, \$8, 0x5555 # zero-extend: 0000aaaa %
2009ffff	% (2c)		addi \$9, \$0, -1 # sign-extend: ffffffff %
312affff	% (30)		andi \$10, \$9, 0xffff # zero-extend:0000ffff %
01493025	% (34)		or \$6, \$10, \$9 # or: ffffffff %
01494026	% (38)		xor \$8, \$10, \$9 # xor: ffff0000 %
01463824	% (3c)		and \$7, \$10, \$6 # and: 0000ffff %
10a00003	% (40)		beq \$5, \$0, shift # if \$5 = 0, goto shift %
00000000	% (44)	ds1ot2:	nop # DS %
08000008	% (48)		j loop2 # jump loop2 %
00000000	% (4c)	ds1ot3:	nop # DS %
2005ffff	% (50)	shift:	addi \$5, \$0, -1 # \$5 = ffffffff %
000543c0	% (54)		sll \$8, \$5, 15 # <<15 = ffff8000 %
00084400	% (58)		sll \$8, \$8, 16 # <<16 = 80000000 %
00084403	% (5c)		sra \$8, \$8, 16 # >>16 = ffff8000 (arith) %
000843c2	% (60)		srl \$8, \$8, 15 # >>15 = 0001ffff (logic) %
08000019	% (64)	finish:	j finish # dead loop %
00000000	% (68)	ds1ot4:	nop # DS %
00004020	% (6c)	sum:	add \$8, \$0, \$0 # sum function entry %
8c890000	% (70)	loop:	lw \$9, 0(\$4) # load data %
01094020	% (74)		add \$8, \$8, \$9 # sum, PIPELINE STALLS %
20a5ffff	% (78)		addi \$5, \$5, -1 # counter - 1 %
14a0fffc	% (7c)		bne \$5, \$0, loop # finish? %
20840004	% (80)	ds1ot5:	addi \$4, \$4, 4 # address + 4, DS %
03e00008	% (84)		jr \$ra # return %
00081000	% (88)	ds1ot6:	sll \$2, \$8, 0 # move result to \$v0, DS %

最右侧的 16 进制数是机器码,直接存放在 inst\_rom.hex 中,被 CPU 读取。左侧的

内容是机器码翻译之后的汇编语言，以及汇编语言的语义（用注释表示）

数据存储：

```
0 : BF800000;      % 1 01111111 00..0 fp -1 %
14 : 000000A3;     % (50) data[0] %
15 : 00000027;     % (54) data[1] %
16 : 00000079;     % (58) data[2] %
17 : 00000115;     % (5C) data[3] %
```

## loop:

首先我们直接将 0 通过 lui 载入到\$1 中，那么\$1 中的值就是 0，之后我们让\$1 和 80 做立即与操作，得到的结果是 80，也是十六进制的 0x50，存入\$4，\$4 存入了 0x50.

之后我们使用 jal sum 语句，进行跳转，注意跳转语句在流水线中使用了延时槽（delay slot, dslot, or DS），他之后的语句 `addi $5, $0, 4` 也会执行，于是\$5 寄存器将存入 0x04，之后到达 6c 行的 add 函数，首先将 0 存入寄存器\$8，之后我们的语句 lw \$9, 0(\$4)，按照\$4 寄存器的值（0x50），索引内存，得到 00000A3，存入\$9，这样一来，\$9 的寄存器将存入 A3，之后的 add 语句表示\$8 需要累加\$9 的值（这些语句都在一个循环体里面），这次是加上 A3，然后\$5 寄存器从 4 开始递减，下一语句 `bne $5, $0, loop` 判断\$5 是否已经递减到零，若尚未，首先要执行延时槽中的指令 `addi $4, $4, 4`，使\$4 寄存器中的值累加，那么载入字的语句将会给\$9 载入内存 0x54 号的值，也就是 0x27，\$8 将累加\$9 的值，此时变成 0xca，依次类推，直到 bne 指令不再跳转，退出 loop:代码段。之后我们会通过 jr 返回到 0x10 行的 return 字段,但是注意这个跳转命令之后还有一个延时槽，使得\$2 的值成为\$8 的值 258.

loop 的工作：

寄存器	操作	1 轮的循环值	2 轮	3 轮	4 轮
\$4	从 0x50 开始按 4 递增	0x50	0x54	0x58	0x5c
\$5	从 4 开始按 1 递减至 0	4	3	2	1
\$8	累加\$9 的值	0xA3	0xCA	0x143	0x258
\$9	读出 data[\$4]的值	0xA3	0x27	0x79	0x115

返回到 0x10 行之后，我们需要执行的语句是 `sw $2, 0($4)`，找到内存的\$4 号位置（0x5c），将\$2 寄存器的内容(0x258)存入，之后的语句 `lw $9, 0($4)`，又将这个值读回来了，读入到\$9 中，这样\$9 的内容成为 0x258,语句 `sub $8, $9, $4`，令  $\$8 = \$9 - \$4 = 0x258 - 0x5c = 0x1f8$ .之后的一个 addi 语句使得\$5 寄存器存入

了 3.

## loop2

之后我们又进入了一个循环 Loop2, 这个循环主要用于测试逻辑运算, 我们首先使得\$5 寄存器的值按 1 递减, 然后让\$5 和 0xffff 按位取或 (ori), 结果(也是 0xffff)存入\$8, \$8 再和立即数 0x5555 取抑或 (xori), 结果(0xaaaa)写回到\$8。

之后我们处理\$9, 初值是\$0 - 1 = -1 = 0xffffffff, \$9 和立即数 0x0000ffff 按位与 (andi)之后得到的结果也是 0x0000ffff, 存入\$10。

然后我们完成 3 步寄存器逻辑运算

1. `or $6, $10, $9`  $\$6 = \$10 \mid \$9 = 0xffff \mid 0xffffffff = 0xffffffff$ ,
2. `xor $8, $10, $9`  $\$8 = \$10 \wedge \$9 = 0xffffffff \wedge 0xffff = 0xffff0000$ ,
3. `and $7, $10, $6`  $\$7 = \$10 \& \$9 = 0xffffffff \& 0xffff = 0x0000ffff$

然后我们判断\$5 是否为 0, 否则重做这个循环。

我们发现, \$5 在递减之后和 0xffff 的按位或操作一定 是 0xffff, 所以\$8 的值每次循环都不变, \$9 的值每次循环都重新计算  $0-1 = -1$  也不变, 于是我们每次循环中, \$8 会先变成 0xffff, 之后保持 0xaaaa, 然后\$9 则一直为 0xffffffff, \$6 一直为 0xffffffff, \$8 一直为 0xffff0000, \$7 一直为 0x0000ffff.

## Shift

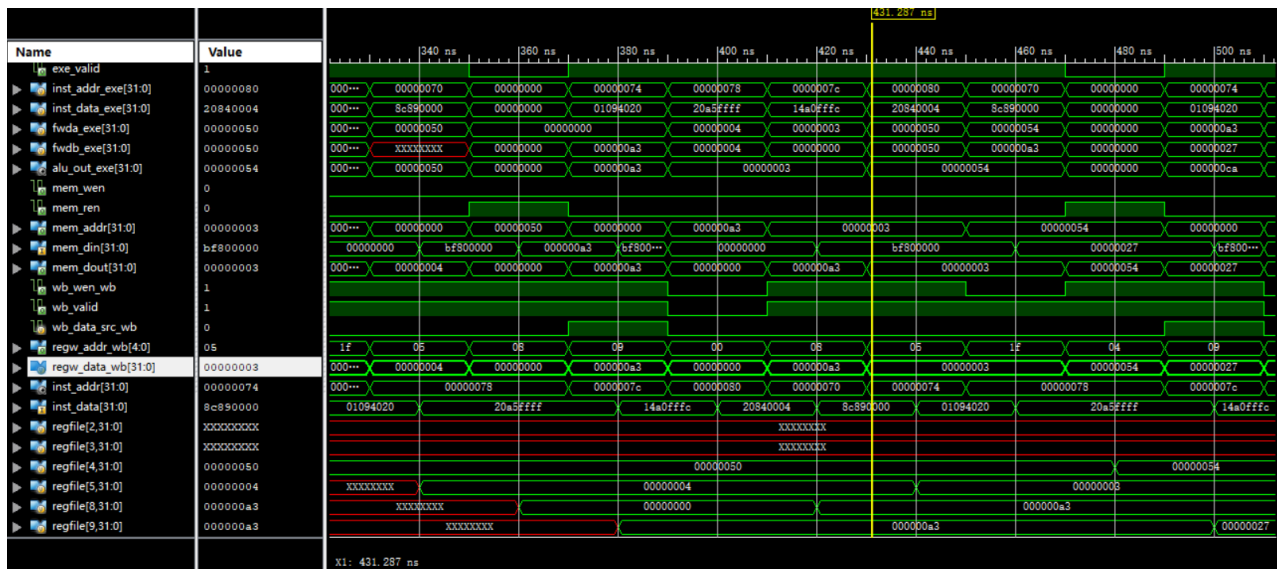
当 5 递减到 0 之后, 我们跳转到 0x50 行的 shift 语句, 这里没有循环, 是测试移位指令的程序, 我们可以将理论结果展示如下: 1.

- |                                   |   |
|-----------------------------------|---|
| 1. <code>addi \$5, \$0, -1</code> | $\$5 = \$0 - 1 = ffffffff$                      |
| 2. <code>sll \$8, \$5, 15</code>  | $\$8 = \$5 \ll 15 = ffffffff \ll 15 = ffff8000$ |
| 3. <code>sll \$8, \$8, 16</code>  | $\$8 = \$8 \ll 16 = ffff8000 \ll 16 = 80000000$ |
| 4. <code>sra \$8, \$8, 16</code>  | $\$8 = \$8 \gg 16 = 80000000 \gg 16 = ffff8000$ |
| 5. <code>srl \$8, \$8, 15</code>  | $\$8 = \$8 \gg 15 = ffff8000 \gg 15 = 0001ffff$ |

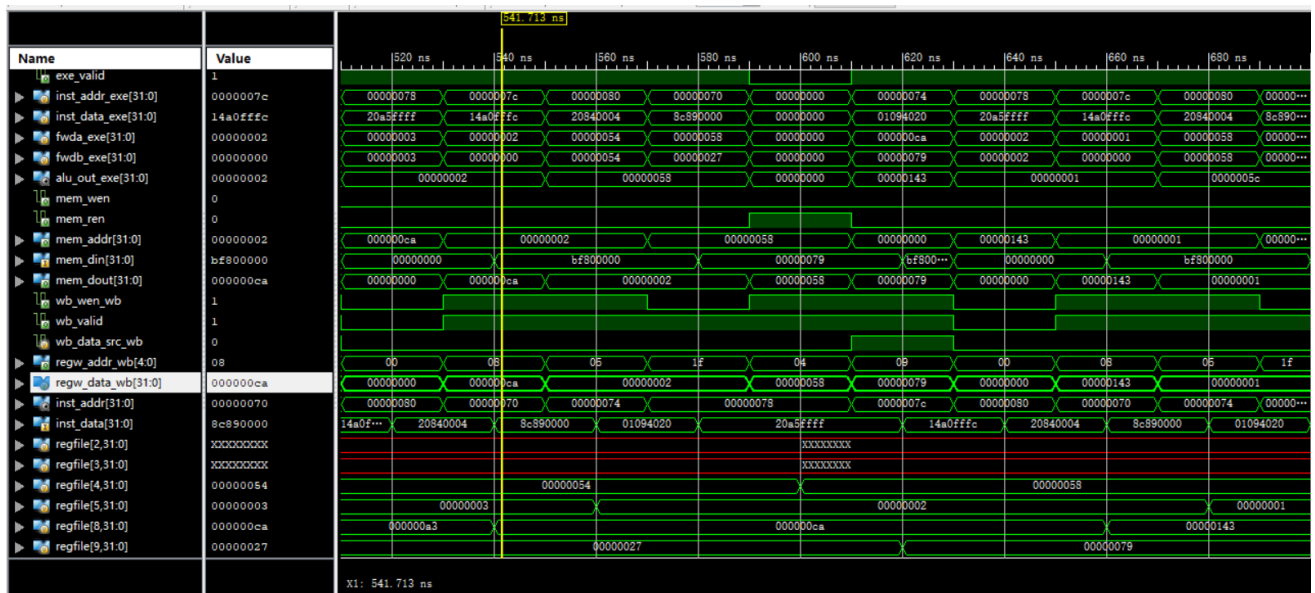
之后就是跳转到自身的 j 指令死循环, 程序结束。

### 4.3 仿真结果展示

开始阶段:

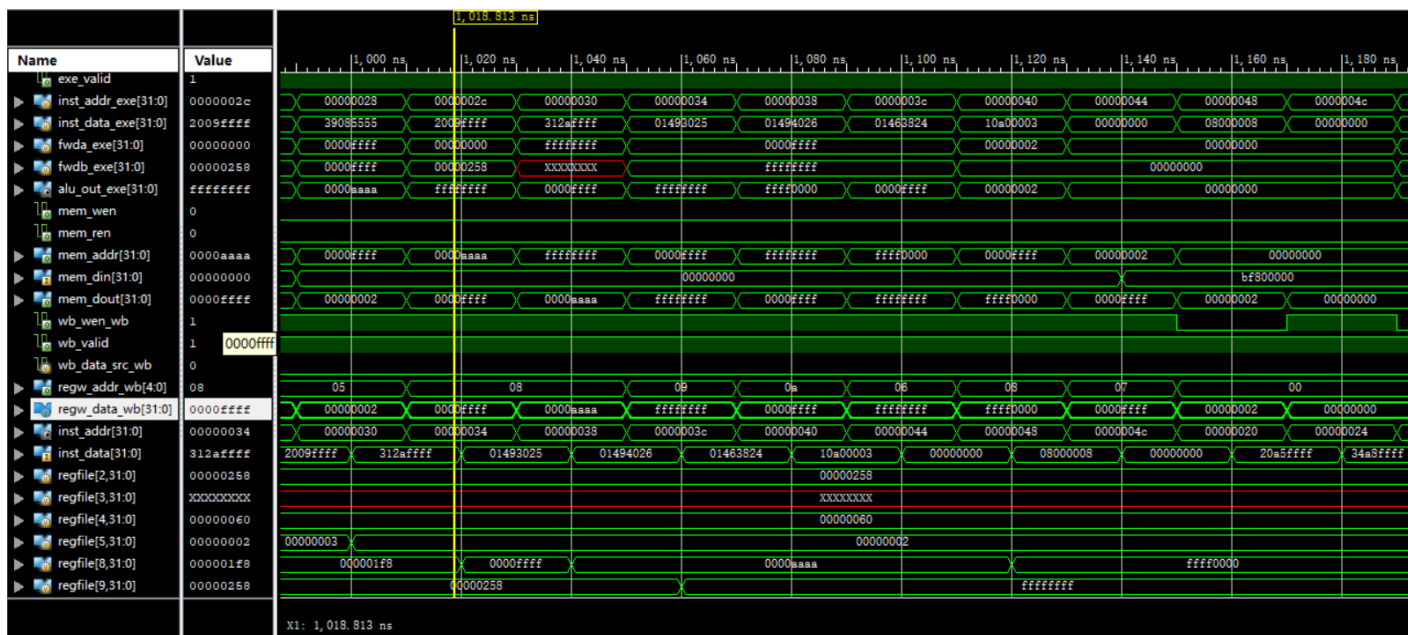


(Loop 循环部分)

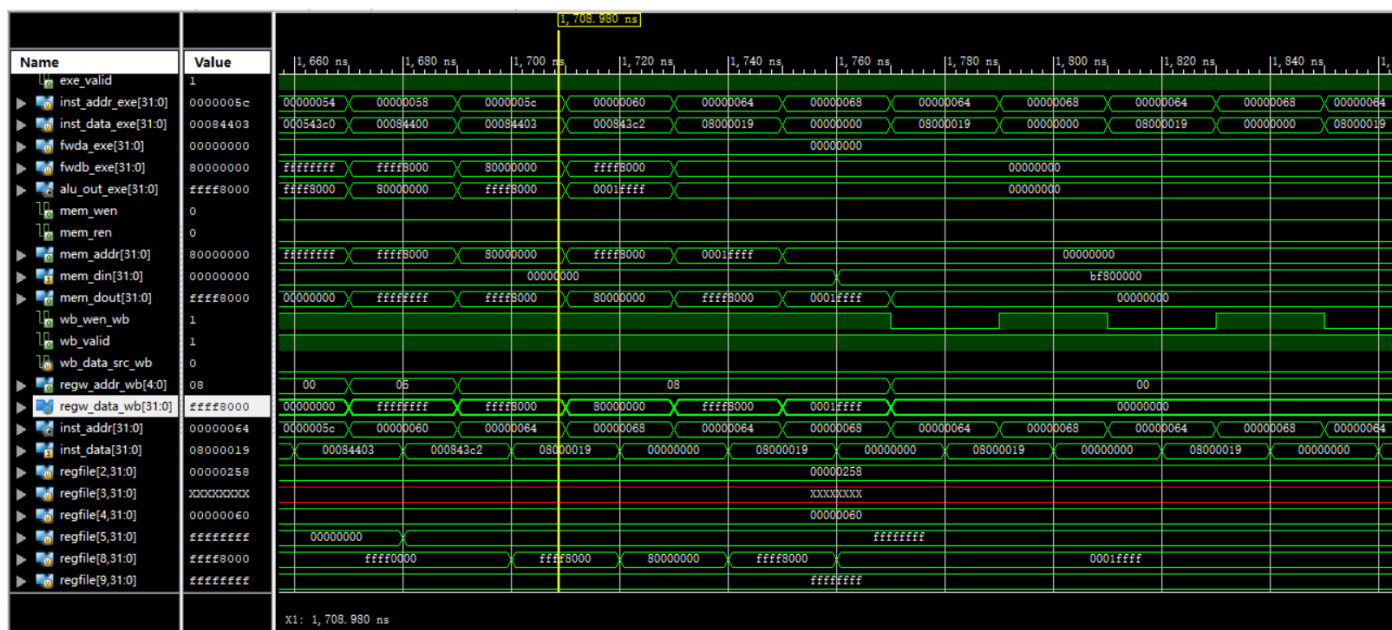


(Loop2 循环部分)



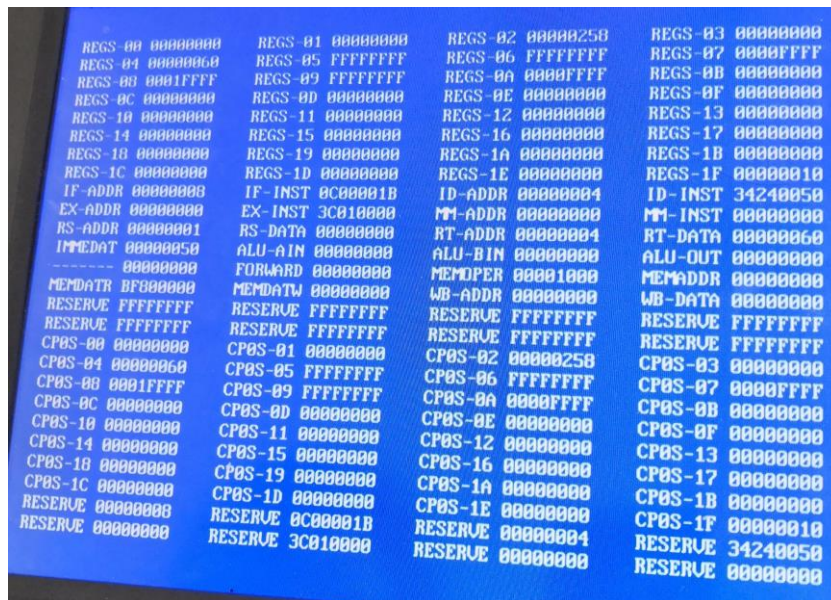


(shift 部分和结尾)



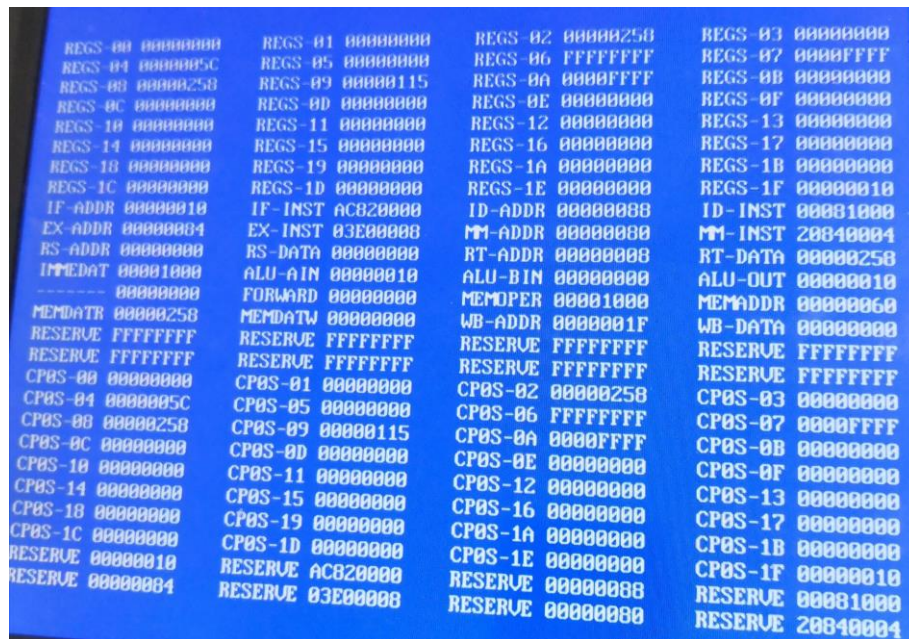
#### 4.4 在 sword 开发板上的仿真结果

因为已经通过现场测试，这里就不再展示全部的情况，仅仅选择部分展示



REGS-00 00000000	REGS-01 00000000	REGS-02 00000250	REGS-03 00000000
REGS-04 00000060	REGS-05 FFFFFFFF	REGS-06 FFFFFFFF	REGS-07 0000FFFF
REGS-08 0001FFFF	REGS-09 FFFFFFFF	REGS-0A 0000FFFF	REGS-0B 00000000
REGS-0C 00000000	REGS-0D 00000000	REGS-0E 00000000	REGS-0F 00000000
REGS-10 00000000	REGS-11 00000000	REGS-12 00000000	REGS-13 00000000
REGS-14 00000000	REGS-15 00000000	REGS-16 00000000	REGS-17 00000000
REGS-18 00000000	REGS-19 00000000	REGS-1A 00000000	REGS-1B 00000000
REGS-1C 00000000	REGS-1D 00000000	REGS-1E 00000000	REGS-1F 00000010
IF-ADDR 00000000	IF-INST 0C00001B	ID-ADDR 00000004	ID-INST 34240050
EX-ADDR 00000000	EX-INST 3C010000	MM-ADDR 00000000	MM-INST 00000000
RS-ADDR 00000001	RS-DATA 00000000	RT-ADDR 00000004	RT-DATA 00000060
IMMEDAT 00000050	ALU-AIN 00000000	ALU-BIN 00000000	ALU-OUT 00000000
----- 00000000	FORWARD 00000000	MEMOPER 00001000	MEMADDR 00000000
MEMDATR BF000000	MEMDATW 00000000	WB-ADDR 00000000	WB-DATA 00000000
RESERVE FFFFFFFF	RESERVE FFFFFFFF	RESERVE FFFFFFFF	RESERVE FFFFFFFF
RESERVE FFFFFFFF	RESERVE FFFFFFFF	RESERVE FFFFFFFF	RESERVE FFFFFFFF
CP0S-00 00000000	CP0S-01 00000000	CP0S-02 00000250	CP0S-03 00000000
CP0S-04 00000060	CP0S-05 FFFFFFFF	CP0S-06 FFFFFFFF	CP0S-07 0000FFFF
CP0S-08 0001FFFF	CP0S-09 FFFFFFFF	CP0S-0A 0000FFFF	CP0S-0B 00000000
CP0S-0C 00000000	CP0S-0D 00000000	CP0S-0E 00000000	CP0S-0F 00000000
CP0S-10 00000000	CP0S-11 00000000	CP0S-12 00000000	CP0S-13 00000000
CP0S-14 00000000	CP0S-15 00000000	CP0S-16 00000000	CP0S-17 00000000
CP0S-18 00000000	CP0S-19 00000000	CP0S-1A 00000000	CP0S-1B 00000000
CP0S-1C 00000000	CP0S-1D 00000000	CP0S-1E 00000000	CP0S-1F 00000010
RESERVE 00000000	RESERVE 0C00001B	RESERVE 00000004	RESERVE 34240050
RESERVE 00000000	RESERVE 3C010000	RESERVE 00000000	RESERVE 00000000

下文节选一些中间过程



REGS-00 00000000	REGS-01 00000000	REGS-02 00000250	REGS-03 00000000
REGS-04 0000005C	REGS-05 00000000	REGS-06 FFFFFFFF	REGS-07 0000FFFF
REGS-08 00000250	REGS-09 00000115	REGS-0A 0000FFFF	REGS-0B 00000000
REGS-0C 00000000	REGS-0D 00000000	REGS-0E 00000000	REGS-0F 00000000
REGS-10 00000000	REGS-11 00000000	REGS-12 00000000	REGS-13 00000000
REGS-14 00000000	REGS-15 00000000	REGS-16 00000000	REGS-17 00000000
REGS-18 00000000	REGS-19 00000000	REGS-1A 00000000	REGS-1B 00000000
REGS-1C 00000000	REGS-1D 00000000	REGS-1E 00000000	REGS-1F 00000010
IF-ADDR 00000010	IF-INST AC820000	ID-ADDR 00000000	ID-INST 00001000
EX-ADDR 00000004	EX-INST 03E00000	MM-ADDR 00000000	MM-INST 20040004
RS-ADDR 00000000	RS-DATA 00000000	RT-ADDR 00000000	RT-DATA 00000250
IMMEDAT 00001000	ALU-AIN 00000010	ALU-BIN 00000000	ALU-OUT 00000010
----- 00000000	FORWARD 00000000	MEMOPER 00001000	MEMADDR 00000060
MEMDATR 00000250	MEMDATW 00000000	WB-ADDR 0000001F	WB-DATA 00000000
RESERVE FFFFFFFF	RESERVE FFFFFFFF	RESERVE FFFFFFFF	RESERVE FFFFFFFF
RESERVE FFFFFFFF	RESERVE FFFFFFFF	RESERVE FFFFFFFF	RESERVE FFFFFFFF
CP0S-00 00000000	CP0S-01 00000000	CP0S-02 00000250	CP0S-03 00000000
CP0S-04 0000005C	CP0S-05 00000000	CP0S-06 FFFFFFFF	CP0S-07 0000FFFF
CP0S-08 00000250	CP0S-09 00000115	CP0S-0A 0000FFFF	CP0S-0B 00000000
CP0S-0C 00000000	CP0S-0D 00000000	CP0S-0E 00000000	CP0S-0F 00000000
CP0S-10 00000000	CP0S-11 00000000	CP0S-12 00000000	CP0S-13 00000000
CP0S-14 00000000	CP0S-15 00000000	CP0S-16 00000000	CP0S-17 00000000
CP0S-18 00000000	CP0S-19 00000000	CP0S-1A 00000000	CP0S-1B 00000000
CP0S-1C 00000000	CP0S-1D 00000000	CP0S-1E 00000000	CP0S-1F 00000010
RESERVE 00000010	RESERVE AC820000	RESERVE 00000000	RESERVE 00001000
RESERVE 00000004	RESERVE 03E00000	RESERVE 00000000	RESERVE 20040004



REGS-00 00000000	REGS-01 00000000	REGS-02 00000258	REGS-03 00000000
REGS-04 00000060	REGS-05 00000000	REGS-06 FFFFFFFF	REGS-07 0000FFFF
REGS-08 000001F8	REGS-09 00000258	REGS-0A 0000FFFF	REGS-0B 00000000
REGS-0C 00000000	REGS-0D 00000000	REGS-0E 00000000	REGS-0F 00000000
REGS-10 00000000	REGS-11 00000000	REGS-12 00000000	REGS-13 00000000
REGS-14 00000000	REGS-15 00000000	REGS-16 00000000	REGS-17 00000000
REGS-18 00000000	REGS-19 00000000	REGS-1A 00000000	REGS-1B 00000000
REGS-1C 00000000	REGS-1D 00000000	REGS-1E 00000000	REGS-1F 00000010
IF-ADDR 00000020	IF-INST 39085555	ID-ADDR 00000024	ID-INST 34A8FFFF
EX-ADDR 00000020	EX-INST 20A5FFFF	MM-ADDR 0000001C	MM-INST 20050003
RS-ADDR 00000005	RS-DATA 00000000	RT-ADDR 00000000	RT-DATA 000001F8
IMMEDAT 0000FFFF	ALU-AIN 00000003	ALU-BIN FFFFFFFF	ALU-OUT 00000002
----- 00000000	FORWARD 00000000	MEMOPER 00001000	MEMADDR 00000003
MEMDATR BF000000	MEMDATW 000001F8	WB-ADDR 00000000	WB-DATA 000001F8
RESERVE FFFFFFFF	RESERVE FFFFFFFF	RESERVE FFFFFFFF	RESERVE FFFFFFFF
RESERVE FFFFFFFF	RESERVE FFFFFFFF	RESERVE FFFFFFFF	RESERVE FFFFFFFF
CP0S-00 00000000	CP0S-01 00000000	CP0S-02 00000258	CP0S-03 00000000
CP0S-04 00000060	CP0S-05 00000000	CP0S-06 FFFFFFFF	CP0S-07 0000FFFF
CP0S-08 000001F8	CP0S-09 00000258	CP0S-0A 0000FFFF	CP0S-0B 00000000
CP0S-0C 00000000	CP0S-0D 00000000	CP0S-0E 00000000	CP0S-0F 00000000
CP0S-10 00000000	CP0S-11 00000000	CP0S-12 00000000	CP0S-13 00000000
CP0S-14 00000000	CP0S-15 00000000	CP0S-16 00000000	CP0S-17 00000000
CP0S-18 00000000	CP0S-19 00000000	CP0S-1A 00000000	CP0S-1B 00000000
CP0S-1C 00000000	CP0S-1D 00000000	CP0S-1E 00000000	CP0S-1F 00000010
RESERVE 00000020	RESERVE 39085555	RESERVE 00000024	RESERVE 34A8FFFF
RESERVE 00000020	RESERVE 20A5FFFF	RESERVE 0000001C	RESERVE 20050003

REGS-00 00000000	REGS-01 00000000	REGS-02 00000258	REGS-03 00000000
REGS-04 00000060	REGS-05 00000002	REGS-06 FFFFFFFF	REGS-07 0000FFFF
REGS-08 FFFF0000	REGS-09 FFFFFFFF	REGS-0A 0000FFFF	REGS-0B 00000000
REGS-0C 00000000	REGS-0D 00000000	REGS-0E 00000000	REGS-0F 00000000
REGS-10 00000000	REGS-11 00000000	REGS-12 00000000	REGS-13 00000000
REGS-14 00000000	REGS-15 00000000	REGS-16 00000000	REGS-17 00000000
REGS-18 00000000	REGS-19 00000000	REGS-1A 00000000	REGS-1B 00000000
REGS-1C 00000000	REGS-1D 00000000	REGS-1E 00000000	REGS-1F 00000010
IF-ADDR 00000020	IF-INST 20A5FFFF	ID-ADDR 0000004C	ID-INST 00000000
EX-ADDR 00000040	EX-INST 00000000	MM-ADDR 00000044	MM-INST 00000000
RS-ADDR 00000000	RS-DATA 00000000	RT-ADDR 00000000	RT-DATA 00000000
IMMEDAT 00000000	ALU-AIN 00000000	ALU-BIN 00000000	ALU-OUT 00000000
----- 00000000	FORWARD 00000000	MEMOPER 00001000	MEMADDR 00000000
MEMDATR BF000000	MEMDATW 00000002	WB-ADDR 00000000	WB-DATA 00000002
RESERVE FFFFFFFF	RESERVE FFFFFFFF	RESERVE FFFFFFFF	RESERVE FFFFFFFF
RESERVE FFFFFFFF	RESERVE FFFFFFFF	RESERVE FFFFFFFF	RESERVE FFFFFFFF
CP0S-00 00000000	CP0S-01 00000000	CP0S-02 00000258	CP0S-03 00000000
CP0S-04 00000060	CP0S-05 00000002	CP0S-06 FFFFFFFF	CP0S-07 0000FFFF
CP0S-08 FFFF0000	CP0S-09 FFFFFFFF	CP0S-0A 0000FFFF	CP0S-0B 00000000
CP0S-0C 00000000	CP0S-0D 00000000	CP0S-0E 00000000	CP0S-0F 00000000
CP0S-10 00000000	CP0S-11 00000000	CP0S-12 00000000	CP0S-13 00000000
CP0S-14 00000000	CP0S-15 00000000	CP0S-16 00000000	CP0S-17 00000000
CP0S-18 00000000	CP0S-19 00000000	CP0S-1A 00000000	CP0S-1B 00000000
CP0S-1C 00000000	CP0S-1D 00000000	CP0S-1E 00000000	CP0S-1F 00000010
RESERVE 00000020	RESERVE 20A5FFFF	RESERVE 0000004C	RESERVE 00000000
RESERVE 00000040	RESERVE 00000000	RESERVE 00000044	RESERVE 00000000

## 五、 讨论与心得

我们在实验四中的遇到的 bug 是目前研究的最久的，而这次实验似乎虽然内容多，但实现起来是比较简单的，老师本来是要求在布置之后的当天验收的，但是因为当时在期中考，所以依旧是两周时间，实验中没有遇到比较严重的 bug，算是幸事一件